Reliable Customized Control Software in an R&D Environment

T.B.H. Kuiper[a], J.G. Leflang[a] and Thang Trinh[b]

[a]Jet Propulsion Laboratory, California Institute of Technology,
4800 Oak Grove Drive, Pasadena, CA 91109, USA

[b]User Technology Associates Inc.,
225 South Lake Avenue, Pasadena, CA 91101, USA

ABSTRACT

Scheduled among the deep space communications activities of the 70-m antennas of the NASA Deep Space Network (DSN) are diverse astronomical observing programs with different requirements. For example, the US Space VLBI Project puts great emphasis on reliability for a few well-defined types of observations, for which the software is essentially frozen for the duration of the Project. On the other hand, Solar System Radar research and observations of regions of star formation need ongoing development, sometimes in real-time, of data acquisition and monitor and control software. This paper describes the methodology by which we can allow each user or project a high degree of customization. To do this we rely on a mixture of public domain software (e.g. Perl, Tcl, Tk, PGPLOT) and locally developed software. The scheme allows the software configuration in the Radio Astronomy Controller to be switched to an observer's or project's specific configuration within minutes, including specific releases of public domain software.

At the core of the Radio Astronomy Controller is a server that controls the R&D equipment. The behavior of this server is largely determined by Tcl scripts, which are customized for the observer or project. An observer working interactively can use a customized Tk client to direct the server via TCP, as well as DSN operational (non-R&D) equipment via another server which communicates with DSN controllers. A project or user may alternatively run a client which is specialized for unattended operations.

Keywords: configuration management, custom software, monitor and control, radio astronomy, Tcl/Tk

1. INTRODUCTION

Radio astronomy observations with the DSN are being automated to simplify operations for DSN personnel, enable remote directing and monitoring by investigators, allow the use of short blocks of antenna time, and use of unanticipated antenna availability on very short notice. We can provide investigators with flexibility or high reliability (but usually not both at the same time).

The DSN comprises three Deep Space Communications (DSC) Complexes — near Canberra, Australia, in Goldstone, California, and near Madrid, Spain. The antennas and standard equipment are nearly identical at the Complexes. R&D equipment varies considerably, but we try to standardize as much as possible to avoid duplication of work.

This paper describes the overall architecture and key elements of the DSN R&D Control System (RDC) designed to achieve these goals.

2. OVERALL ARCHITECTURE

2.1. Objectives

2.1.1. Non-interference with tracking operations

Radio astronomy in the DSN involves a mix of configuration-controlled equipment used in routine deep space communications and R&D equipment which is regularly reconfigured to meet experiment goals. R&D equipment and experiments may not put routine operations at risk.

Other author information: (Send correspondence to T.B.H.K)
T.B.H.K.: JPL mail-stop 169-506; E-mail: kuiper@jpl.nasa.gov
J.G.L.: JPL mail-stop 301-235; E-mail: jleflang@tmod.jpl.nasa.gov
T.T.: JPL mail-stop 301-235; E-mail: thang@zebra.jpl.nasa.gov

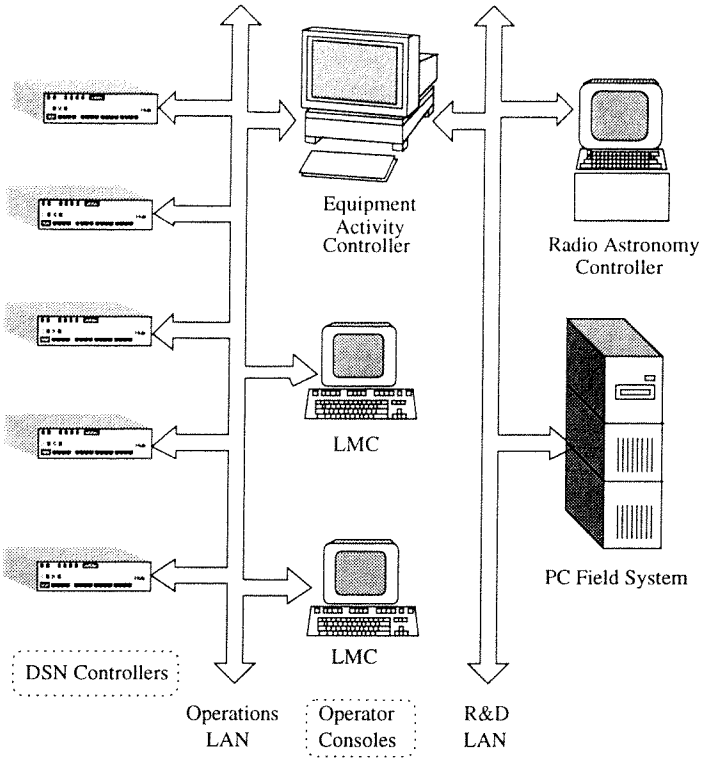

Figure 1. Overall architecture of the DSN R&D control system. A DSN operator normally uses a Link Monitor Console (LMC) to direct tracking operations. The Equipment Activity Controller (EAC) provides similar capability for R&D operations, and also controls R&D equipment, some examples of which are shown connected to the R&D LAN.

2.1.2. Remote operation and monitoring

Working within the DSN tracking schedule, radio astronomy is normally assigned single sessions separated by several to many days, so that travel to a DSN site for the purpose of observations is inconvenient at best. Efficiency of radio astronomy experiments is greatly enhanced by remote operation.

2.1.3. Unattended observations

Even more efficient use can be made of available blocks of antenna time if the investigator does not have to participate in the actual data acquisition. Projects involving extensive mapping or long integrations are well suited to automated, unattended data acquisition.

2.2. Approach

Figure 1 shows the overall configuration. The Equipment Activity Controller (EAC) provides the DSN operator with control over both DSN standard and R&D equipment. The EAC functions as a DSN operator console. Like a standard Link Monitor Console (LMC), it is connected to the Complex's operations LAN and is able to control and monitor DSN equipment controllers. Unlike the LMCs, it is also connected to the R&D LAN, and is able to control and monitor R&D equipment controllers. The EAC graphical user interface (GUI) normally runs on the EAC, but need not, allowing for remote operation.

Messages between the client and servers are Tcl (1) commands extended with the Extended Tcl (TclX) toolkit (2). Messages are passed using a simple TCP/IP protocol called Net Services, which is described in Sec. 4. The Tcl command set has been augmented with Net Services commands to allow rapid development of Tcl/Tk clients.

Figure 2 shows the tasks which run in the EAC to support an R&D session. xant provides the operator's GUI. oci provides a command line interface by which the operator can issue commands to DSN standard subsystems for

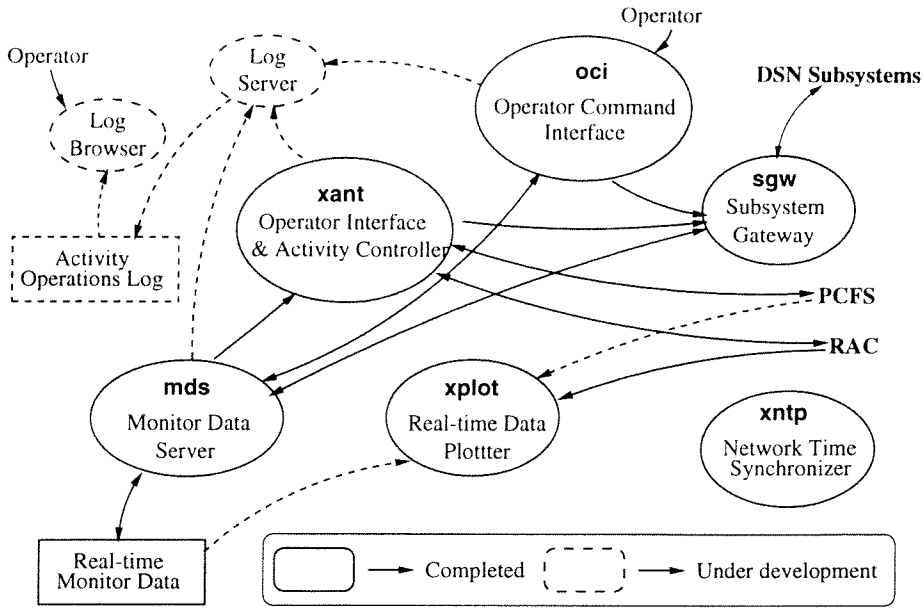

Figure 2. Schematic of the EAC software. xant provides the operator's GUI. Multiple xant's can be used to control different antennas.

operations which are not yet handled by xant. All communications with DSN standard subsystems go through sgw, the subsystem gateway.

In an interactive session directed by a DSN operator (or an investigator sitting with the DSN operator), the EAC is a client of both the DSN controllers and the R&D controllers, which function as servers. (A *client* makes requests for services. A *server* controls resources and provides services.) However, while maintaining its client aspect with respect to the DSN servers, the EAC can also act as a server to clients on the R&D LAN. The EAC will accept commands from both the PCFS and RAC, enabling either of those to be the focus of the experiment, with the EAC acting effectively as a server providing access to DSN antennas and receivers. The design also allows a user-developed program (e.g. a Tk script) on a remote computer (e.g. at JPL) to be the focus of the experiment. For example, during a VLBI pass, the EAC functions as a server to the PC Field System, which directs the activities.

All communications with equipment not physically connected to the DSN's operational network are secured through the use of hardware encryption units. For all practical purposes, such equipment is part of the DSN closed network. Encryption units allow the closed network to be flexibly and rapidly extended using open network segments.

3. R&D EQUIPMENT CONTROL

3.1. Objective

The diverse R&D activities in the DSN have different requirements for equipment control. Rather than trying to develop a common software suite to meet all needs, we thought it preferable to allow each activity to have its own, potentially highly customized control software. The method for reliably invoking the correct software is discussed in Sec. 5.

3.2. Approach

Some equipment have their own, specialized controllers, whose functionality is closely tied to the capabilities of the equipment. Other than the ability to communicate via Net Services messages (see Sec. 5), such controllers do not need to be versatile.

Most of the general radio astronomy and other R&D equipment (such as the Goldstone Solar System Radar) is controlled by the Radio Astronomy Controller (RAC). There are two RACs (one serving as hot back-up) at each DSC

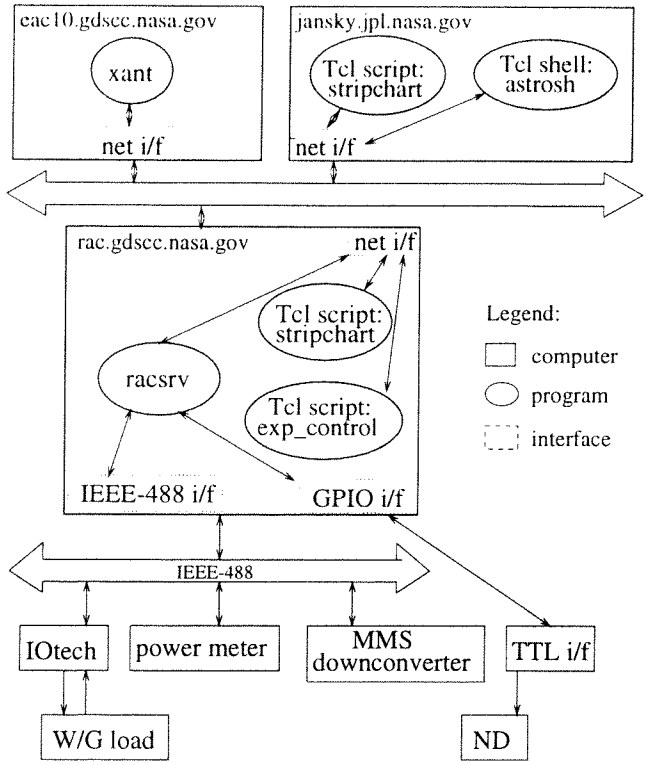

Figure 3. An example of the processes involved in a typical R&D session.

Complex. They are either Hewlett Packard 9000 model 735s or model C180s. The RAC provides both the physical interfaces and the software by which these interfaces are managed. The instrumentation software interface is through HP's Standard Instrument Control Library, which we have found to be robust for the IEEE-488 interface bus. The RAC runs a server program called racsrv which provides access to the radio astronomy equipment. Even when a client running on the RAC is the central focus of an experiment, this client communicates with racsrv, not directly with the equipment (see Fig. 3). racsrv services requests from local and remote clients via network connections. Net Services messages conform to a particular, but quite flexible, format (see Sec. 4).

To achieve the desired versatility, racsrv has almost all of its functionality defined by Tcl scripts, which can be easily tailored to individual needs. The core of racsrv is the select loop. It monitors a socket for new connection requests, the sockets of current connections, and a pipe from Timer Services. Messages that a timer has expired causes the main Tcl interpreter to source the timer's Tcl script file. These script files can be edited while racsrv is running, so that the user can substantially modify the behaviour of racsrv in real time. Each connected client gets its own interpreter, so that the various users are isolated from each other. All clients have access to a set of shared variables. Further details on racsr are given in Appendix A.

Only one client have access to the instrument control command toolkit, either one of the tasks known internally to racsrv as EAC and CTL. Only the controlling client is able to change the shared variables. A "pass control" mechanism is achieved by giving control to the first EAC or CTL tasks that connects to racsrv. To pass control to another client, the controlling client disconnects, enabling the next EAC or CTL task that connects to take control. For example, this might be used when the EAC xant is the main controlling client, temporarily passing control to a CTL client for diagnosing R&D hardware problems.

4. INTERPROCESS COMMUNICATION

4.1. Objective

Our highly distributed control system requires a flexible, easy to use interprocess communications services.

4.2. Approach

Net Services provides application programs running under UNIX (and, for historical reasons, VxWorks) with a simple, socket-based networking and inter-process communication facility. These services allow processes to communicate within a single processor, between processors across a backplane, or between processors over a network. They also allow processes to communicate with each other in any combination. In all cases, the services look identical to the application programs. The application programming interface (API) for network I/O that basically looks similar to that for file I/O. The typical sequence of calls is represented in Table 1.

Table 1. A representative sequence of Net Services calls

Server	Client	Description
net_init		listen for connection requests
	net_connect	request a connection
net_accept		establish a connection
net_getpeername		get the host name and process name
	net_send	send a command message
net_recv		receive a command message
net_send		send a response message
	net_recv	receive a response message
	net_close	close the connection
net_close		close the connection

The API maintains the file descriptor semantics of UNIX to allow application processes to perform network I/O multiplexing via the "select" system call. This is particularly useful when an application process may be waiting for connection requests from more than one endpoint, or when data may arrive from many input sources, possibly together with other connection requests.

The services are based on Internet domain sockets and support the TCP/IP protocol. Since TCP is a byte-stream protocol that does not provide any record boundaries to the communication data stream, the API for the network services will support a message-based service that preserves the sender's message boundaries for the receiver process. Messages can be sent and received as discrete units between application programs regardless of the protocols being used, thereby simplifying their communication interface.

Appendix A shows how racsrv uses Net Services. Details on the Net Services message structure are presented in Appendix B.

5. SOFTWARE CONFIGURATION MANAGEMENT

5.1. Objectives

5.1.1. Versatility

It is unusual for the radio astronomy equipment to be used for the same project for more than antenna session at a time. Typically, each session using R&D equipment will require reconfiguration of the software. (Hardware reconfiguration is generally done under software control and is therefore not discussed here.)

5.1.2. Reliability

Some projects, such as the US Space VLBI Project, pus great emphasis on reliability for a few well-defined types of observations, for which the software is essentially frozen for the duration of the project. Such projects require that changes made by other users will not affect their operations.

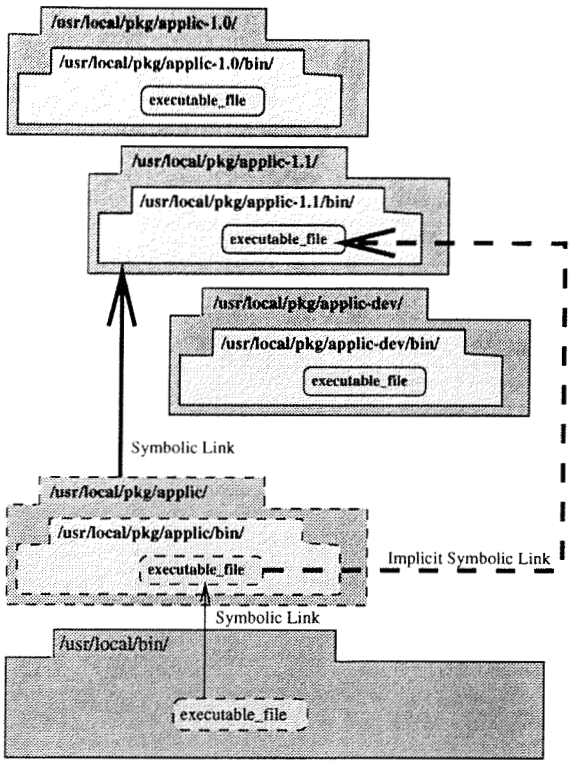

Figure 4. Example of the method for selecting the files from a specific version of a software package Items labels with solid lines are real entities. Dashed lines indicate virtual entities. The file labelled executable_file shows one instance of a file which is globally available to users who have /usr/local/bin/ in their PATH.

5.2. Approach

There are a number of version control systems available under UNIX, including Concurrent Versions System[3,4] (CVS), Revision Control System[5–7] (RCS) and Source Code Control System (SCCS). These could be used to manage locally developed software but are not suitable for managing multiple versions of contributed software packages. Systems for managing packages include Depot,[8] based on the Andrew File System (AFS), and Working Version File System,[9] based on the Network File System (NFS). Both these systems maintain unions of versions of packages, which optimizes the use of disk space. However, removing redundancy between versions of packages also entails greater risk, in that damage to a key file could affect many versions. The Debian/GNU Linux method of package management[10] keeps versions intact, but changing versions is time consuming since entire files are moved during the installation process. Also, this system has not been implemented under HP-UX (used on the RACs) or Solaris (used on the EACs).

5.2.1. Software package selection

UNIX expects to find various components of the software in conventional places. For example, executables will be in directories specified by the users environment variable PATH, and typically includes the directories /bin, /usr/bin, /usr/local/bin, etc. Likewise, run-time libraries and scripts are expected to be in /lib, /usr/lib, /usr/local/lib, etc. Similarly, man documentation is expected in certain directories. Instead of putting our files in these directories, we put there instead links to the actual files. That allows us to keep all the files for a given software package together in one directory tree.

Because in general there are multiple versions of a package, we use another link to point to the specific version of the package. For example, /usr/local/bin/tcl is a pointer to /usr/local/pkg/tclX/bin/tcl. However, /usr/local/pkg/tclX is actually a pointer to, say, /usr/local/pkg/tclX-7.4.

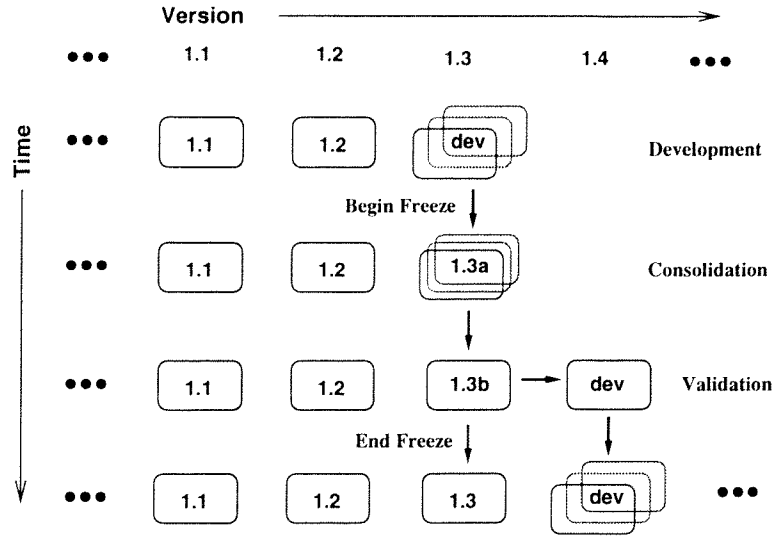

Figure 5. Example of one cycle in the development process of local packages.

By our convention, every numbered version of a package is a frozen version. The only non-frozen version has the suffix dev instead of a version number. In general, an operational software version is a frozen version, unless the investigator specifically requests the current development version.

At the start of a session, the operator will (probably without being aware of it, or at least the details) run the pkg_select script to set the correct links for all the necessary applications and packages. The script also selects all the packages on which the named package depends, so only the highest level package(s) need to be selected. pkg_select and other pkg commands are described in 5.2.3 below.

5.2.2. Software package development

Software development and maintenance is done at three sites, as well as at JPL. Freezing software which has been under active development at multiple sites adds the challenge of consolidating the various versions. During the α-phase the current version is consolidated. Local modifications are collected and integrated at a central site for that package. (There is no reason for all packages to be consolidated at the same site.) The consolidated version is returned to the sites for testing and this is iterated until the α-version runs without obvious problems at all sites. It then becaomes a β-version. After a suitable period of stable performance, it is frozen and the β label is removed. Figure 5 illustrates the development cycle.

5.2.3. Package management commands

Table 2 lists the commands we created to manage software packages. To prevent a random user from inadvertently changing the package selection, only members of the group ops can execute the command pkg_select. Even the owner of the file and root cannot execute this command, because the command itself checks the group of the user.

6. CONCLUSION

Many details have necessarily been omitted from this paper, but are available at http://DSNra.jpl.nasa.gov/devel/.

The main area of current EAC software development is automation for configuring and monitoring DSN antenna and standard equipment. The goal is to alleviate the DSN operators from mundane tasks, and to make remote and unattended operations more secure.

RAC software development is now concentrating on providing customized software for the various radio astronomy projects, with emphasis on automation.

Table 2. Package Management Commands

Permissions			Group	Command	Function
Owner	Group	Other		**Package Selection**	
rwx	r-x	r–	ops	pkg_admin	Install or remove one package
rwx	r-x	r–	ops	pkg_clr_all	Clear all packages. If done as root, it will make a clean system for package selections by removing all files with names that conflict with package files.
rwx	r-x	r–	ops	pkg_select	Select a package and all the packages on which it depends and do any special things not done by pkg_admin
				Package Maintenance	
rwx	r-x	r–	ops	pkg_admin	Build a package's links table
rwx	r-x	r–	any	pkg_changes	Report any changes to files in packages since a specified date/time. Options: -h Help; no check done -o file Output; default: stdout -r file Use date/time of this file -t time "Mon Day HH:MM Year"; will prompt if not specified -t over-rides -r
rwx	r-x	r–	any	pkg_check	Check for changes in pkg since the last time the user ran pkg_check. Before using for the first time, user must do "touch PKG_checked" in user's home directory. Results are mailed to user. This is really intended to be run with cron.
rwx	r–	r–	any	pkg_html_index	Make !Index.html for a directory; this version has file: URLs for access using Netscape's "Open file..."
rwx	r–	r–	any	pkg_mk_html	Convert a packages man pages to HTML
rwx	r–	r–	any	pkg_mv_mann	Move man n pages to another section
rwx	r–	r–	any	pkg_publish	Make index.html which is a copy of !Index.html but with http: links for Web server use.
rwx	r-x	r-	any	pkg_updates	Create /usr/local/pkg_update.tar with any files in packages changed since /usr/local/pkg.tar was made or gotten.
				Miscellaneous	
rwx	r-x	r-x	ops	pkg_status	Show the current package selection

APPENDIX A. RACSRV RADIO ASTRONOMY CONTROLLER SERVER

A.1. Purpose

This program listens continuously on a set of sockets, using 'select' to determine which sockets require attention. In addition, it monitors a pipe which is used to notify that one of several possible timers has expired and executes the appropriate Tcl script. Figure 2 summarizes the EAC software.

A.2. Files

The program expects to use the following files with Tcl scripts in /usr/local/lib/racsrv.PROJ/ where PROJ is a project or user specific suffix:

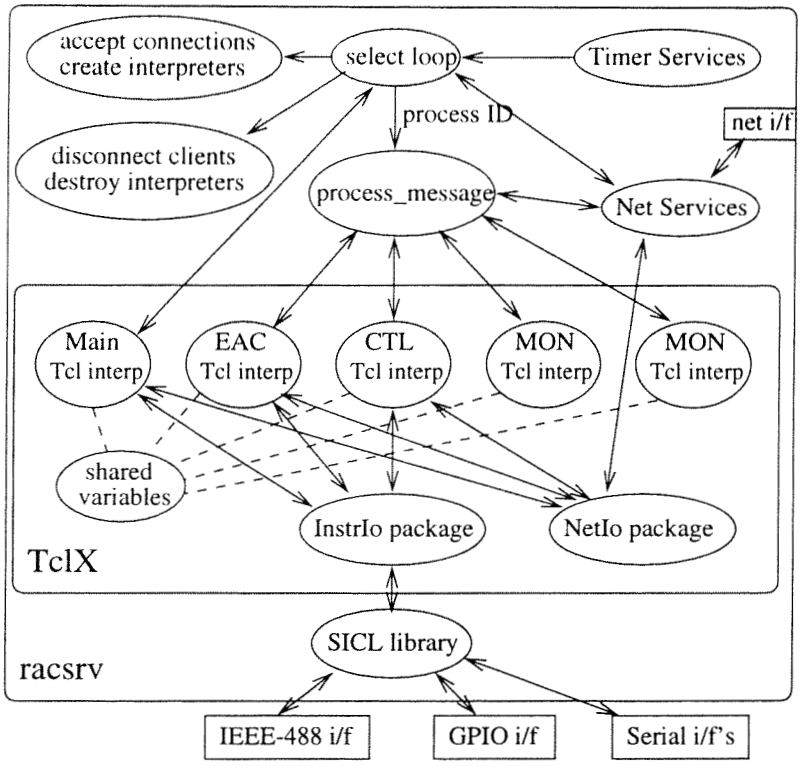

Figure 6. Overview of the racsrv program.

racsrvrc	This script is executed when racsrv starts up.
rac_monrc	This is a start-up Tcl script which is executed whenever a new monitor connection is accepted.
rac_ctlrc	This is a start-up Tcl script which is executed whenever a new controller connection is accepted.
timer_1sec	This is Tcl script which is executed whenever the one second timer expires.
timer_3sec	This is Tcl script which is executed whenever the three second timer expires.
timer_10sec	This is Tcl script which is executed whenever the ten second timer expires.
racsrv_exit	This script is executed when racsrv is terminated with a HUP.

If PROJ is not specified as an argument when startinging racsrv, the suffix dflt will be used. Alternatively, the scripts may be found in /usr/local/pkg/ANY_DIR-VER/lib/racsrv.PROJ/ The above scripts may in turn source other scripts.

The program expects to find the station DSS number defined as a Tcl command in /usr/local/lib/config/this_station.tcl

A.3. Overview

The basic procedure in the main loop is as follows:

- Set the bits in readfds for all the sockets of interest.
  - call FD_ZERO to clear readfds
  - loop over FD_SET for all sockets of interest
- Call select which returns with the bits set for all the sockets that have data.
- Loop over FD_ISSET to process the tasks for all the pending ports.

Initially, the only sockets which are active are

0 stdin
1 stdout
2 stderr
3 listenfd, the one on which new connections are accepted.

If select shows data at the listenfd socket, client_accept() is invoked. In client_accept(), net_accept() establishes a TCP connection and assigns a file descriptor. Then, net_getpeername() is used to identify the client process. If the client process cannot be identified, it is assumed to be an anonymous process (ANY_TASK).

If the client process is identifiable (i.e. pname != 0), its fd is entered in the array client_fd[] at index pname, and is assigned its own Tcl interpreter interp[pname]. If it is not identifiable (pname = 0), its fd is entered in the array anon_client_fd[] at the first available position, and it is assigned a coresponding Tcl interpreter.

After processing a new connection request, all the sockets which have data pending are processed. Any non-responsive socket, or one which returns an error from process_message(), is closed.

APPENDIX B. MESSAGE STRUCTURE

B.1. Message Structure

- **message header** - 8 bytes consisting of
  - **message ID** - 4 bytes consisting of
    - **subsystem ID** - 2 bytes with the numeric code identifying the subsystem originating the message. For the RDC software we have defined
      - EAC_ID - Equipment Activity Controller
      - PCFS_ID - PC Field System
      - RAC_ID - Radio Astronomy Controller
      - RMON_ID - remote monitor subsystem
      - RCTL_ID - remote control subsystem
    - **message code** - 2 bytes with a numeric code consisting of
      - **message length** - one byte, so that the number of 128-byte blocks in the message body is 1 plus the integer value of this byte
      - **message type** - one byte which specifies the message type
  - **source ID** - 4 bytes with the numeric code representing the originating process. For the RDC programs, we have
    - SGW_SRV - SPC LAN gateway server
    - SGW_TASK - SPC LAN gateway server acting as a client (in the EAC)
    - MON_SRV - EAC monitor data server
    - MON_TASK - EAC monitor data server acting as a client (in the EAC)
    - ANT_TASKx (x = 1..8) - antenna control tasks (usually in the EAC)
    - ANT_SRVx (x = 1..8) - antenna control tasks acting as servers
    - CMD_TASK - operator command input task (in the EAC)
    - PCFS_TASK - PCFS client
    - RCTL_TASK - remote control client
    - RAC_SRV - RAC server
    - RAC_TASK - RAC server acting as a client
    - ANY_TASK - anonymous (monitor only) client
- **message body**

B.2. Notes

Messages formats are defined uniquely for each pair of communicating subsystem types in the overall network. For example, there is a unique set of messages defined for PCFS/EAC communications. Note that "subsystems" are not necessarily separate nodes, although they generally are. For example, a client which controls the RAC server (RCTL_ID) often resides on the same node as the server (RAC_ID) itself.

Note that the message types are defined according to the communicating *subsystems*, not the communicating *processes*. This enables various processes on the same subsystem to use the same message types.However, specific message types may have an implicit process dependency. For example, (..CMD) messages go from the client to the server, and responses (..RSP) go from the server to the client.

The length, and potentially even the structure of a message body, for each message can be defined uniquely. Only the sending process and the receiving process need to agree on this. Currently, all command and response messages have the same structure.

ACKNOWLEDGMENTS

Paul Harbison and Charles Naudet developed many of the Tcl/Tk scripts used to operate the R&D equipment. Edward King developed the Perl script pkg_admin.

REFERENCES

1. J. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, Reading, MA, 1994.
2. M. Harrison, *Tcl/Tk Tools*, O'Reilly, Sebastopol, CA, 1997.
3. T. Morse, "Version control: Beyond rcs," *Linux J.* **21**, pp. 43–46, 1996.
4. T. Morse, "Version control: Beyond rcs." http://www.linuxjournal.com/issue21/1118.html.
5. A. Robbins, "What's gnu rcs - revision control system," *Linux J.* **10**, 1995.
6. A. Robbins, "What's gnu rcs - revision control system." http://www.ssc.com/lj/issue10/gnu10.html.
7. M. Welsh and L. Kaufman, *Running Linux*, pp. 383–386. O'Reilly, Sebastopol, CA, 1995.
8. "The depot configuration management project." http://andrew2.andrew.cmu.edu/depot/depot.html.
9. "Working version." http://www.wv.com/.
10. D. Sheetz, *The Debian Linux User's Guide*, Linux Press, Penngrove, CA, 1997.